# Tutorial on I/O workload characterization in MPI applications

# IISWC 2014

Yushu Yao

National Energy Research Scientific Computing Center

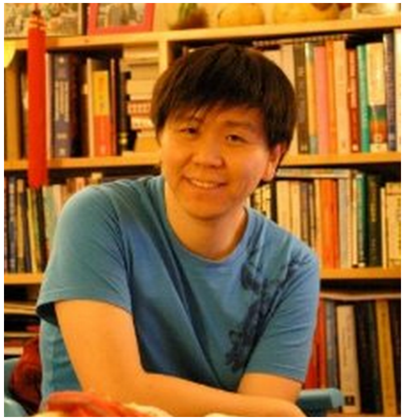
Lawrence Berkeley National Laboratory

Phil Carns

Mathematics and Computer Science Division

Argonne National Laboratory

# Presenters



Yushu Yao

NERSC/LBNL

Phil Carns

ANL

# Plans for Today

- **Basics of Parallel I/O (20′)**
- I/O Performance Characterization and Darshan (30′)
- Typical I/O Bloopers (20′)
- Break / Account Setup (20′)
- Hands-on Exercises (70′)

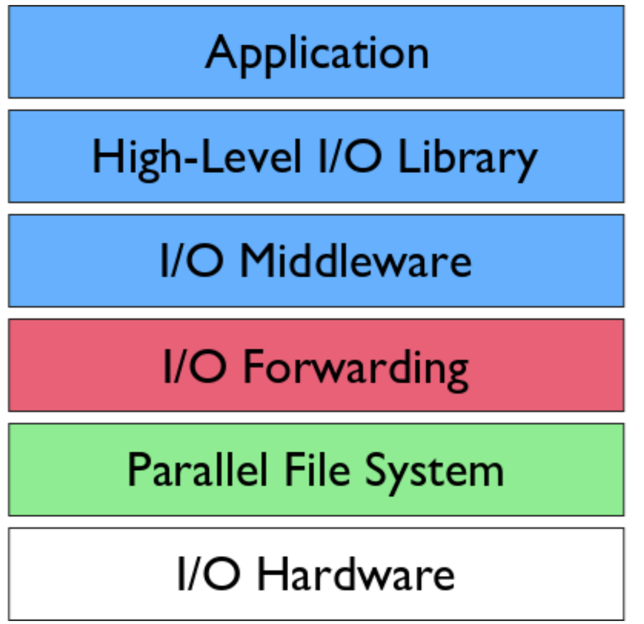
# I/O for Computational Science

**High-Level I/O Library**
maps application abstractions onto storage abstractions and provides data portability.

*HDF5, Parallel netCDF, ADIOS*

**I/O Forwarding**
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

*IBM ciod, IOFSL, Cray DVS*




**I/O Middleware**
organizes accesses from many processes, especially those using collective I/O.

*MPI-IO*

**Parallel File System**
maintains logical space and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*

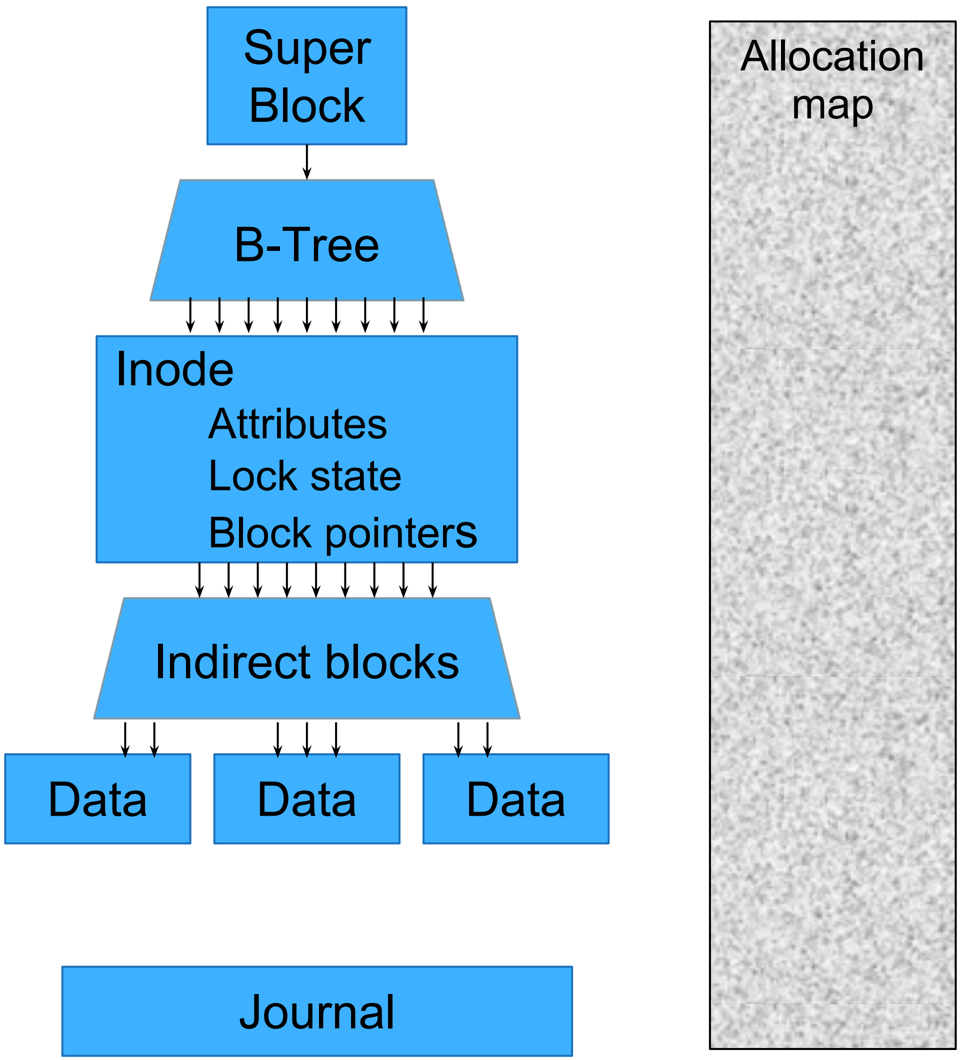
**Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.**

# Local File Systems

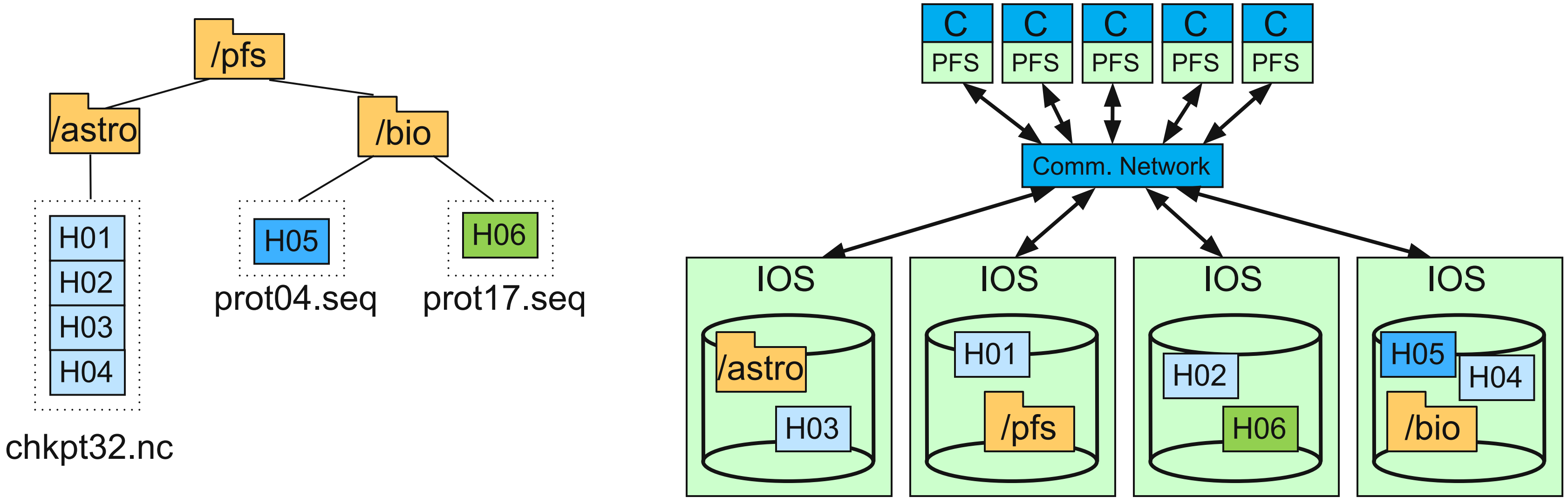
Persistent data structure maps from a user's concept of a file to the data and attributes for that file.

Early research and differentiation was all about optimizing access to a single device

UFS, EXT4, ZFS, NTFS, XFS and BtrFS are local file systems



# Parallel File Systems

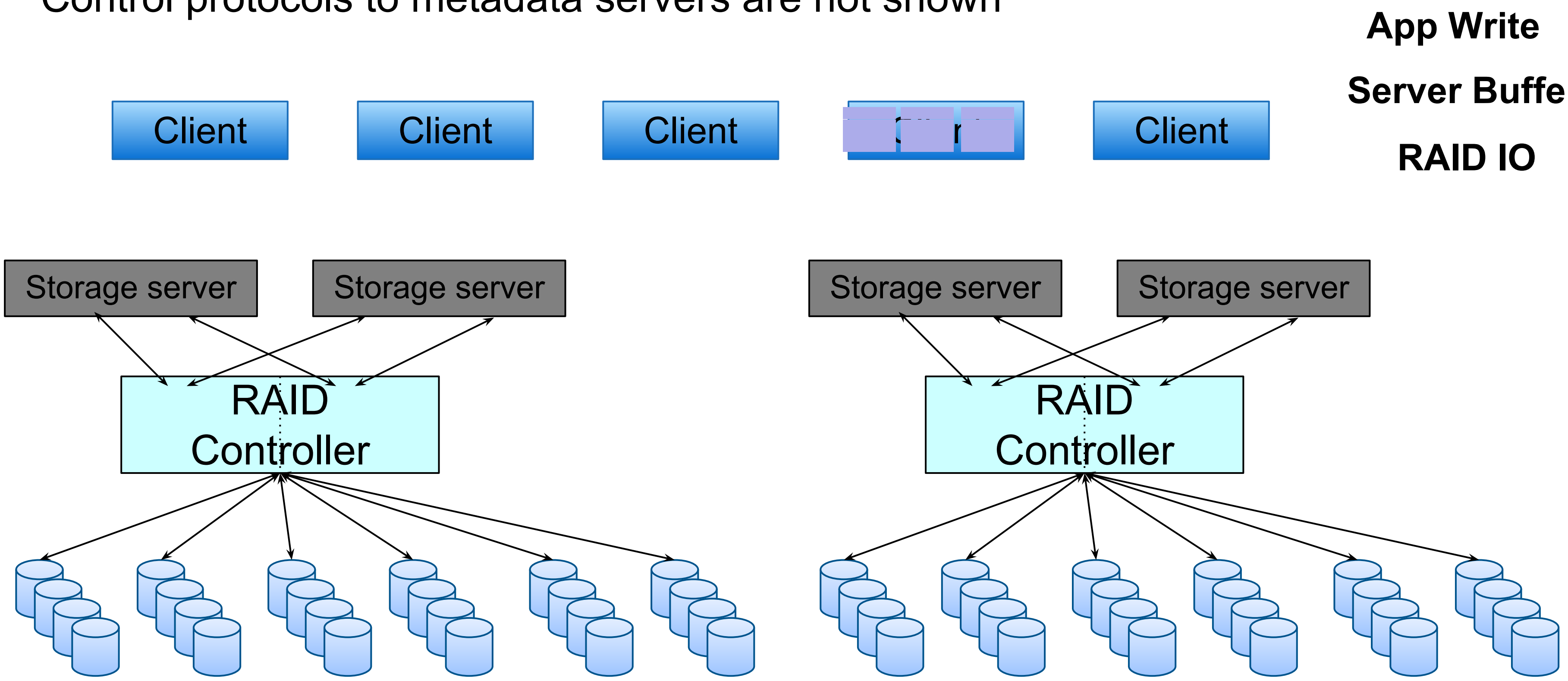



An example parallel file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS) while small bioinformatics files are each stored on a single IOS

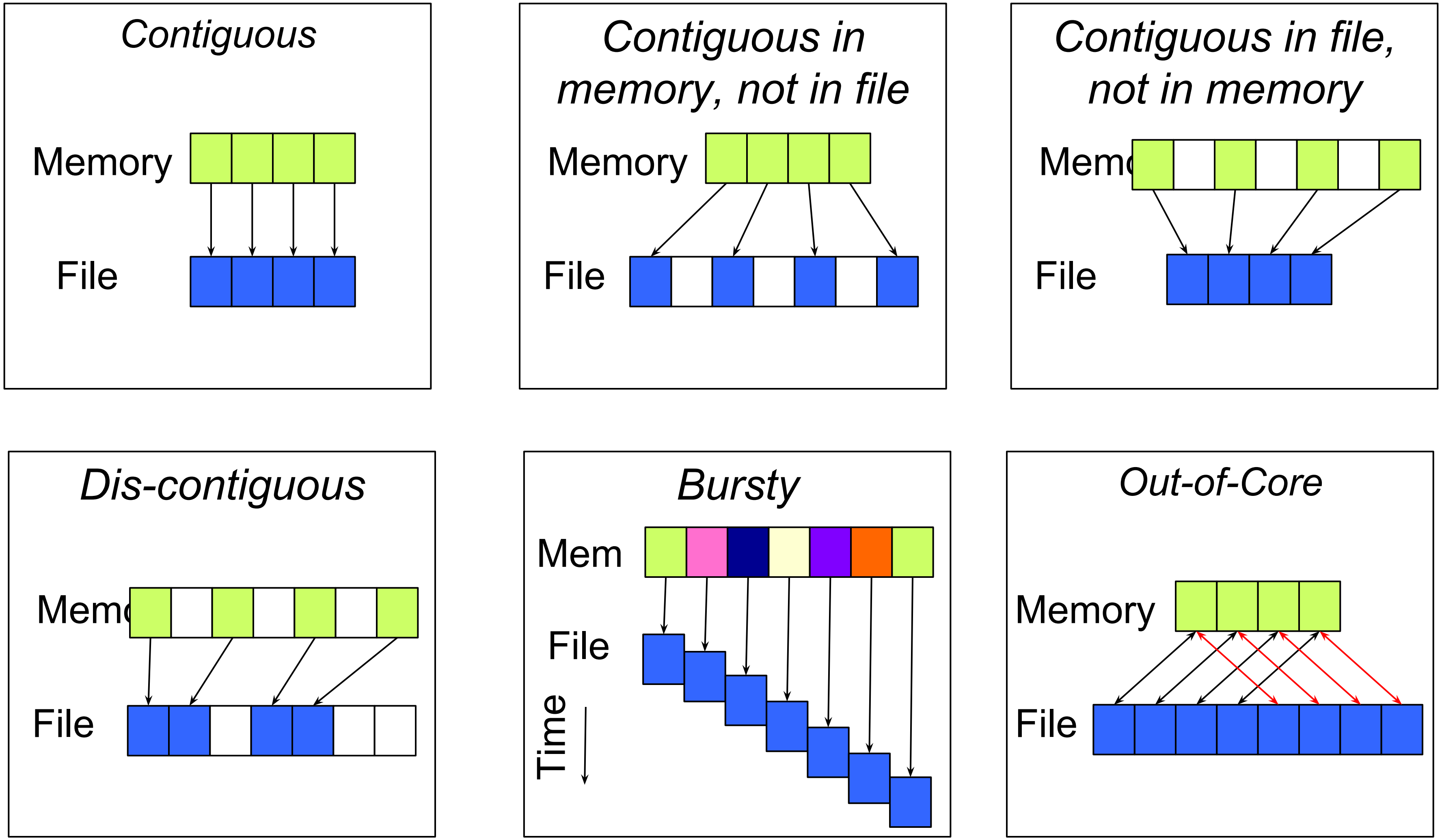
# Lustre and GPFS Data Path

Lustre clients stripe data across Object Storage Servers (OSS), which in turn write data through a RAID controller to Object Storage Targets (OST). OST hides local file system data structures

GPFS has different metadata model but a similar data path
Control protocols to metadata servers are not shown

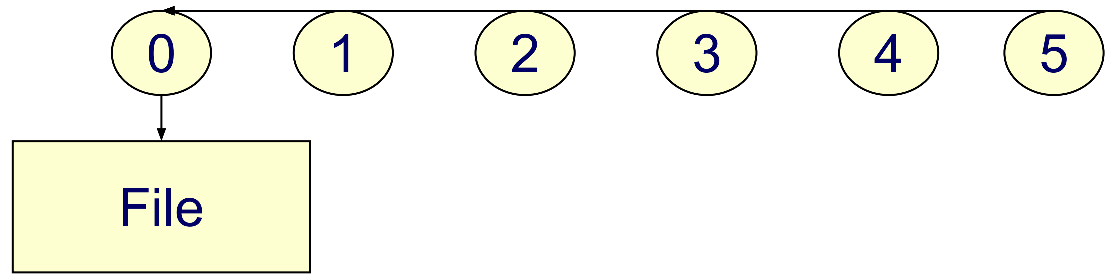


# Access Patterns

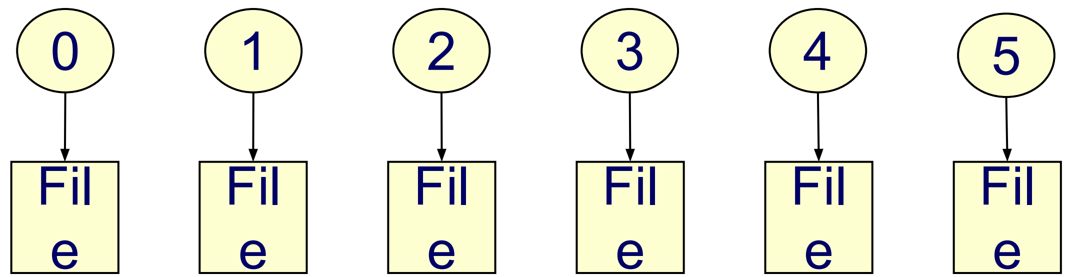


# Serial, multi-file parallel and shared file parallel I/O

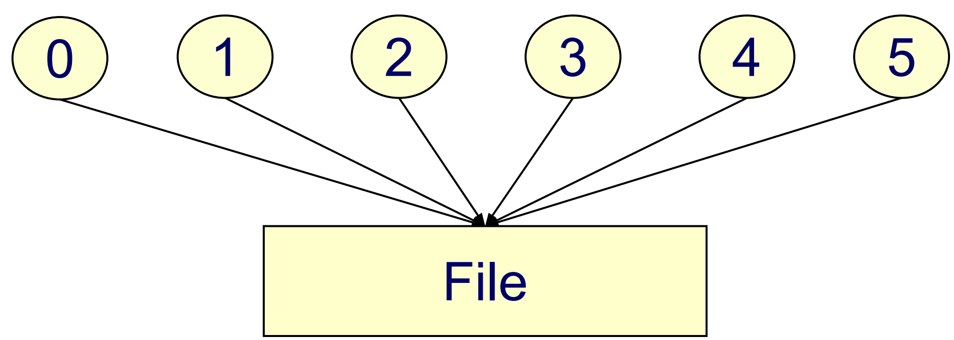
**Serial I/O**




**Parallel Multi-file I/O**




**Parallel Shared-file I/O**



# What's wrong with POSIX?

- It's a useful, ubiquitous interface for basic I/O
- It lacks constructs useful for parallel I/O
  - Cluster application is really one program running on N nodes, but looks like N programs to the filesystem
  - No support for noncontiguous I/O
  - No hinting/prefetching
- Its rules hurt performance for parallel apps
  - Atomic writes, read-after-write consistency
  - Attribute freshness
- POSIX should not have to be used (directly) in parallel applications that want good performance
  - But developers use it anyway

# MPI-IO

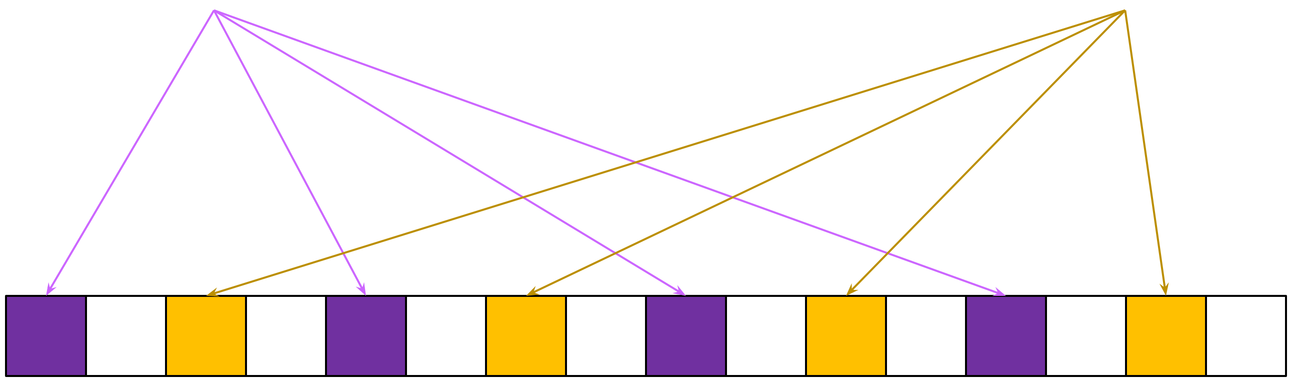
- I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable format (external32)
    - Not self-describing - just a well-defined encoding of types
- Implementations available on most platforms (more later)

# Simple MPI-IO

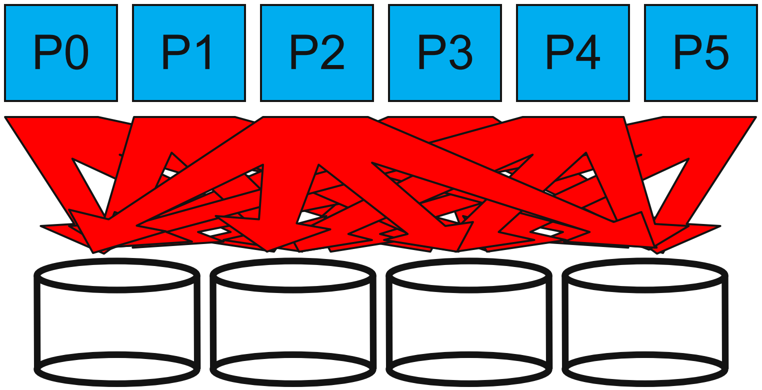
- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype*passedto write

```
MPI_File_open(COMM, name, mode,
info, fh);
MPI_File_set_view(fh, disp, etype,
filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
datatype, status);
```

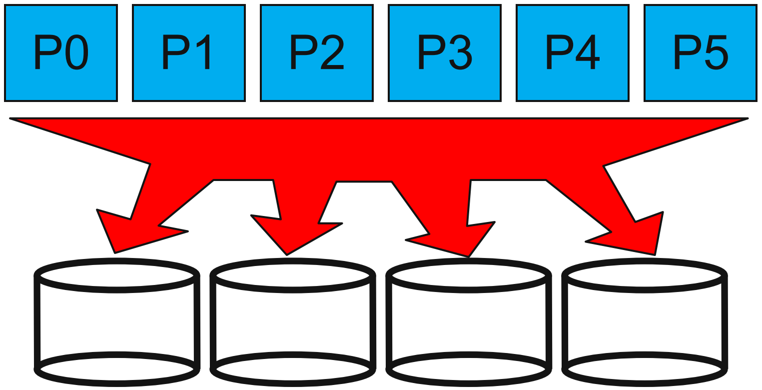
```
MPI_File_open(COMM, name, mode,
    info, fh);
MPI_File_set_view(fh, disp, etype,
filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
datatype, status);
```



# Independent and Collective I/O



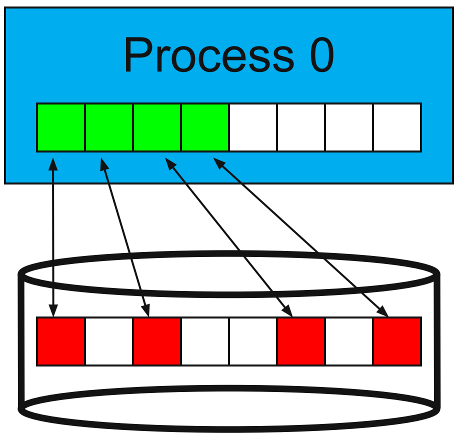

Independent I/O



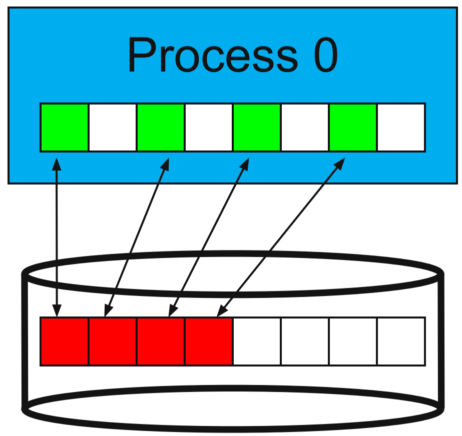

Collective I/O

- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance

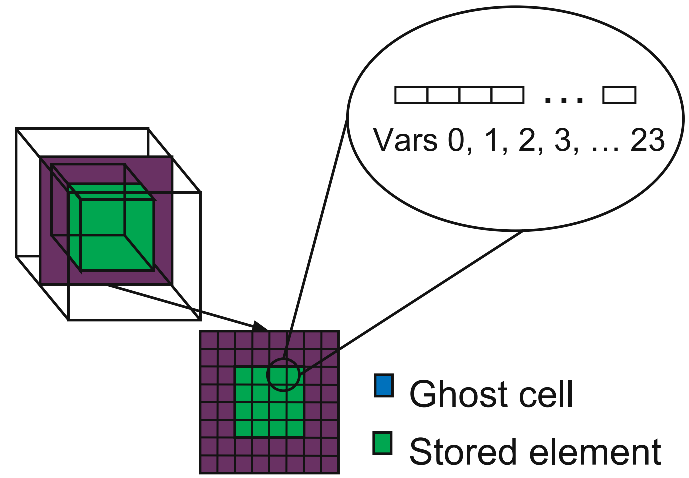
# Contiguous and Noncontiguous I/O




Noncontiguous
in File




Noncontiguous
in Memory




Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O.

- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  – Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

# MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations that result in better I/O performance
- Ideal location in software stack for file system specific quirks or optimizations
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

# General Principles for Better I/O

- Bigger IO Size
- Do not stat
- Try to avoid POSIX shared file
- Minimize Seeking
- Use Collectives when possible
- Use High Level Libraries (HDF5, etc) when possible